1998

Bond Objects - a white paper

Ladislau Bölöni

Report Number:
98-002

Bond Objects -- a white paper

Ladislau Bölöni
Purdue University

CSD-TR-98-002
February 1998

Bond Objects - a white paper

Ladislau Bölöni

February 19, 1998

Abstract

This white paper defines the structure of Bond objects and introduces the *shadows*, an abstraction for networking. In the Bond system shadows are used to implement interobject communication, data transfer, remote execution and various distributed services. Locally managed collection of shadows are used to implement virtual networks. The communication of Bond objects is based on a message passing model, using the KQML agent communication language.

1 Introduction

Distributed systems are built from a collection of standard objects like control agents (scheduler, dispatcher, monitor), execution agents, servers and various types of data [2, 10].

The Bond system is a collection of loosely connected communicating objects of various complexity which provides a framework for distributed systems. Bond uses persistent objects, some related to active agents (representing permanent or temporary services) others being passive objects (internal or external data).

A developer can build a distributed system using Bond objects as building blocks with no concern for the low level details of the system.

The Bond system is written in Java, which provides the advantage of portability. For various services the Bond system relies on the underlying middleware which can be CORBA [6], Infospheres [3, 4], MPI [14], Java RMI [15], HTTP [13]. Bond agents use the KQML [11, 12] agent communication language, which allows Bond objects to implement complex behavior without relying heavily on the middleware. KQML also allows Bond agents to communicate with other agents outside the Bond system.

The communication model of the Bond system is based on three high level functions: message passing, object replication and directory service.

The communication model of the Bond system is based on three high level functions: message passing, object replication and directory service.

Message passing. Bond objects communicate using string messages. Messages are used to implement remote data access and remote procedure calls. Every message is a sentence in the KQML [11] agent communication language.

Message passing is done at the level of local objects using the `say()` function. Communicating between objects is implemented using the *shadow object mechanism.* A shadow object is a local placeholder for a remote object. A message to a remote object is a local function call applied to the shadow of the remote object. Remote message passing happens only between the original object and its shadow.

Object replication. An identical copy of a remote object can be obtained by calling the function `realize()` on the shadow. The replica can be accessed locally, while the master copy can be updated calling the `update()` function on the local copy.

Object migration can be realized using the `migrate()` function on a local copy. This makes the local object the master copy, and target of further updates.

Directory service. The role of the directory service in the Bond system is to create a local shadow of a remote object. The object can be uniquely specified by its name, or can be specified by a query (for example requesting "a scheduler agent").

Shadow objects represent a higher level abstraction for networking. They support (a) messaging, transmission of string messages among objects, (b) directory services and c) remote instantiation of objects. Shadow object relay on a transport mechanism which can be provided by a middleware layer or by transport protocols. The table 1 presents the way in which different transport protocols and middleware systems implement the requirements of the Bond communication model. If a certain service is not provided by the transport mechanism, it should be implemented at the level of Bond shadows.

One of the advantages of using the shadow abstraction is that the only component of the Bond object system which depends on the underlying middleware or transport protocol is the Bond shadow. Porting a Bond networking solution to a new middleware implies only reimplementing the

bondShadow object, while all the other components can be reused even without being recompiled.

For efficiency reasons two Bond objects may choose to communicate directly using the middleware. For example two agents may decide to use a simple ftp protocol for transfering data. This requires a good coordination between the design of the two agents. The message passing interface however, represents a common denominator of all Bond objects, necessary when a Bond object should interact with unknown objects. For example a monitor agent should be prepared to monitor the execution of any Bond agent, without knowing the details of structure.

Middleware	Message passing	Remote object instantiation	Directory service	Overhead
TCP/IP	no	no	no	minimal
HTTP	no	no	no	minimal
MPI	yes	no	no	small
Java RMI	yes	yes	no	small
Infospheres	yes	no	no	large
CORBA	yes	yes	yes	large

Table 1: The capabilities and the overhead of various communication protocols and middleware systems. A "no" entry implies that the service should be implemented by the shadow object

2 The Bond object hierarchy

The Bond objects can be grouped in a hierarchy of objects. The lower in the hierarchy, the more complex is an object. The top of the Bond object hierarchy is presented in the Figure 1.

bondObject it is the root of the hierarchy. It implements the common fields of all Bond objects (name, unique identifier, address and type). It also implements the messaging function say, permitting any Bond object to communicate with any other Bond object using KQML.

bondData is the common ancestor for the objects which represents persistent data in the external storage.

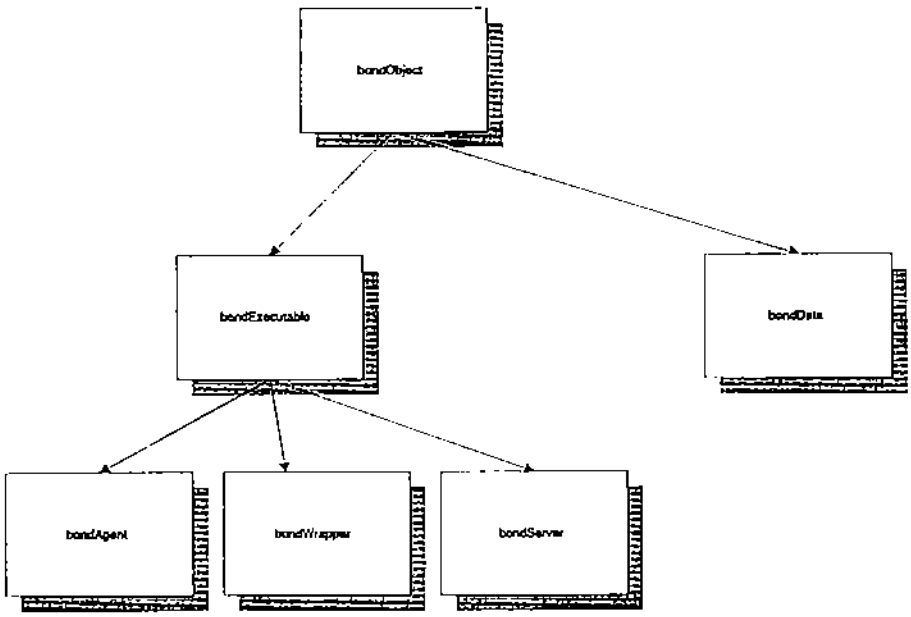

Figure 1: The top of the Bond Object hierarchy

bondExecutable it is the common ancestor of the classes which implement executable programs. A `bondExecutable` has at least one active thread and implements specific functions like start, stop, abort and the corresponding KQML messages.

bondServer represents a Bond server program. Examples include the directory server, the dispatcher and the persistent storage server.

bondWrapper it represents a Bond agent which acts as an interface to an external, non-Bond program (usually a legacy application).

bondAgent represents the common ancestor of the Bond control agents like `Scheduler`, `Dispatcher` and `Monitor`.

3 The structure of a Bond Object

Bond Objects are special cases of Java objects. All of them are descendents of the `bondObject` class. Bond objects can be used to implement data structures in programs as any other object. In addition Bond objects provide

dynamic extensibility of the data structure, communication and migration functions.

My name is Object, Bond Object. Any Bond object inherits four data fields from the `bondObject` class. These fields are needed by the Bond system to identify the object and maintain the consistency.

name The name of the object.

bondID The unique identifier of the object. It is automatically generated when the object is registered.

type The type of the object.

address The internet address of the object. The format of the address depends on the middleware used.

Static fields are implemented as regular fields of the object. They can be accessed directly (without performance penalties), but they should be declared during compile time. They can be also accessed by name using the `get()` and `put()` functions and their messaging equivalents.

The following example show the various ways of accessing the fast-access fields of a Bond object.

```
class MyBondObject
extends bondObject {
  float sfield;
}

sfield = 0.1;
Float temp = this.get("sfield");
this.put("sfield",4.5);
```

Dynamic fields are created during runtime, using the `put()` function. They are implemented as an internal hashtable in the Bond object. They can not be accessed directly.

The following example shows the creation and usage of a dynamic field:

```
MyBondObject mbo = new MyBondObject();
mbo.put("newfield",7); // a new, integer field is created
```

```
System.out.println(mbo.get("newfield");
// ok, 7 will be printed
System.out.println(mbo.get("newfield"));
System.out.println(mbo.newfield);
// compile error: only fast access fields can be used this way
System.out.println(mbo.get("field"));
// runtime exception raised: no such field
```

Accessing fields using messages Both dynamic and fast access fields can be accessed using the KQML equivalents of get and put.

```
(ask-one :content get :param1 newfield)
(tell :content set :param1 newfield :value 9)
```

4 Shadow objects

The shadow objects implement communication and remote access in the Bond system. A shadow object is a local placeholder of a remote object. Any message passed to the shadow object will reach the original object.

Shadow objects are the conceptual equivalents of the stubs in Corba and RMI and they can be implemented using stubs. However, bondShadow is a unique object which represent interface to Bond objects of different types. This eliminate the need for interface description languages and additional compilation steps. Any Bond object can communicate and interactively discover their interfaces.

We illustrate in a simple example the use of the shadow objects. Let us assume that we want to access a remote object called MyObject. First we need a shadow of the object, which we can obtain using the directory object:

```
bondShadow shadow = dir.find("MyObject");
```

If the object is not found an exception will be thrown which should be captured. Now we can use the shadow to send a message to the object.

```
shadow.say("(tell :content get :bondVarname type)", this);
```

This message asks the object about the value of the variable type. The variable type is a common variable for every Bond object so we can be confident that the object will understand our message. As a reply the remote

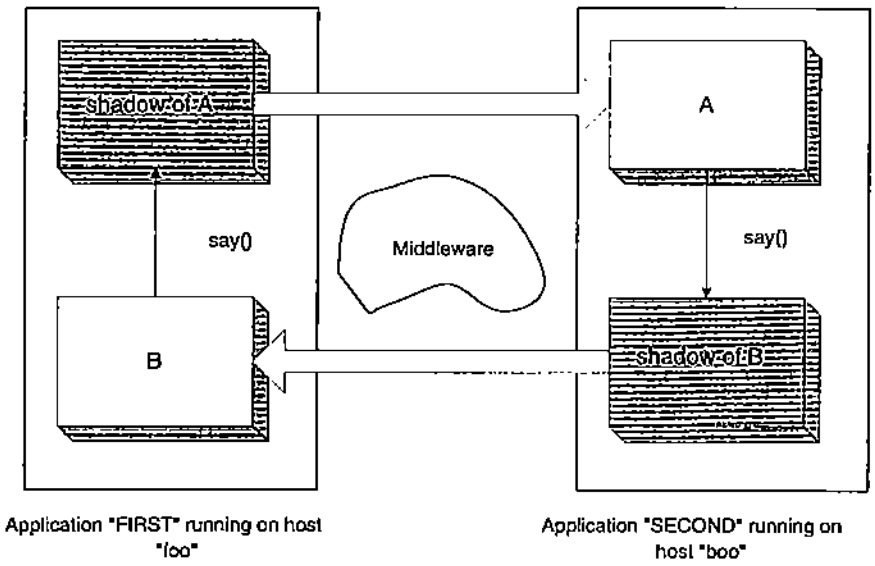

Figure 2: Message passing between two Bond objects using shadows

object sends a message to our local object. To accomplish this, it should create a local shadow of the object as presented in Figure 2. The message sent will be a call to the say function of the local object, and will be something like:

```
say("(tell :content value :sender MyObject :bondVarname type
      :bondValue bondData)", this);
```

Now we know that our object is a bondData object. Accessing objects using KQML messages may be costly if the object contains large amounts of data. One possibility is to create a local copy of the object. This operation is called the *realization* of a shadow.

```
bondData localcopy = shadow.realize();
```

There may be an arbitrary number of replicas of an object, each connected by shadows to the *master copy*. The middleware should assure the consistency between the master copy and the other copies.

We call *migration* the process under which a different copy of the object becomes the master copy.

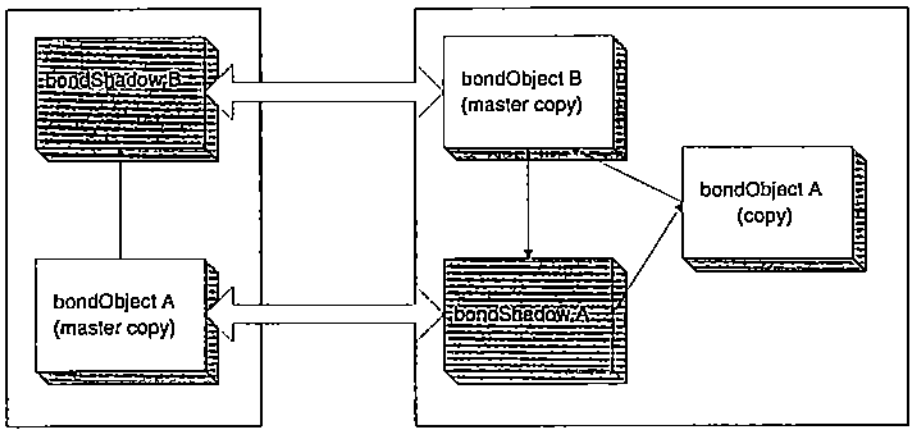

Figure 3: Creating a local copy of a remote Bond Object using realize()

4.1 Virtual networks

A *virtual network* in the Bond system is a local collection of shadow objects. An object creates a virtual network in order to manage a set of semantically related remote objects. Virtual networks can be dynamically created, expanded or deleted. For example the central directory maintains a virtual network of local directories or a scheduler agent maintains a virtual network of the execution agents.

In the following we present a simple application of virtual networks in the execution of a metaapplication in the Bond system. A typical metaapplication has the following life cycle:

Step 1: The execution of the distributed application is initiated by a human user or an agent. The description of the application is passed to the Bond server in a `bondContract` object.

Step 2: The Bond server starts up a number of control agents to supervise the execution of the contract. Depending of the nature of the contract more or less control agents may be needed (see [2]). The control agents link themselves into a virtual network, each control agent creating the shadows of the agents it will communicate with.

Step 3: The scheduler agent contacts the hosts on which the processes will be executed and starts up the corresponding execution agents.

Step 4: The contract is executed by the execution agents under the control of the scheduler and the other control agents.

Step 5: After the successfull execution of the contract, the control

agents are shut down using the information present in the virtual network.

Step 6: The results of the execution are passed to the initiator agent by the Bond server.

From this life cycle one can see that every component distributed application is communicating with a limited number of remote objects. During the startup phase of the application the objects are linked together in a *virtual network*. In the Bond system a virtual network is represented by a collection of Bond shadows.

Observation: the restricted set of objects needed by a component is called *Infosphere*. One can see the virtual network as an internal representation of the infosphere.

4.2 Additional functions of shadows

The bondShadow objects are lightweight objects, usually we consider them as a communication channel. They can incorporate additional functions like message buffering, data caching, local response and security features which will be discussed below.

Message buffering. The shadow objects perform buffering as a default. A message send (say function) applied to the shadow will never fail. The shadow object will buffer all the incoming messages and tries to send them when possible. There is a timeout associated with the shadow object. After the timeout is expired, the shadow object will send the sender object a message specifying that the message is not deliverable.

Short (several seconds) timeout times are useful for hiding the latency of the network. Long (up to several days) timeout times make the shadow objects act as replacement agents for remote objects. For example if the remote object is a human operator which can go away, the shadow object can collect the messages arrived, and deliver them when the operator is back. This function may

Data cacheing. The bondShadow object may incorporate an internal object of time bondCache which contains a partial copy of the data fields of the main object together with a bitmap. The write-through and write-back cache protocols are implemented. Data cachign in a distributed system raises difficult consistency problems. Our current approach is that caching is an explicit decision and responsability of the user.

```
bondShadow bs = dir.find("remote_data_object");
```

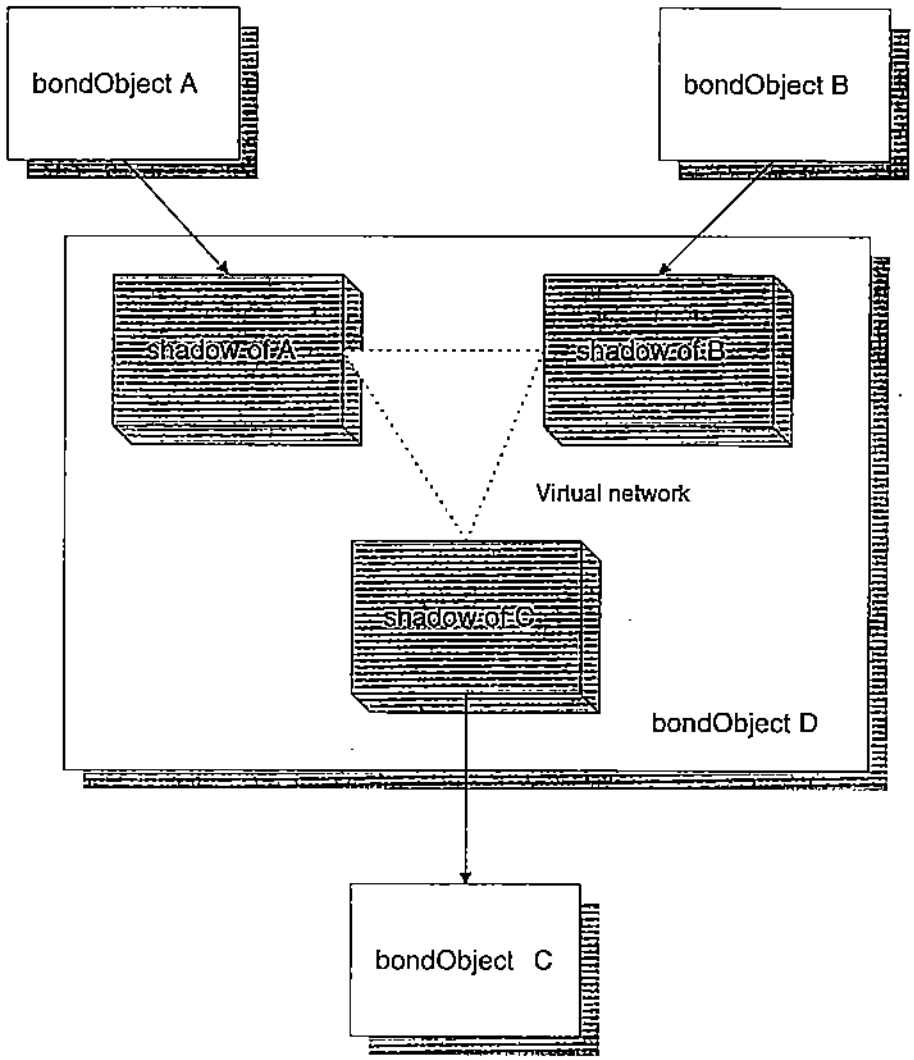

Figure 4: Creating a virtual network using a collection of shadows

```
bs.enableCache(bondCache.WRITETHROUGH);
bs.get(value); // first access, the value will be
bs.get(value); // second access, the data will come from the cache
bs.set(value, 5); // simultaneous updateing of the cache and the object
bs.disableCache();
// no caching will be performed from now on. No updating is needed
```

```
// because of the write-through protocol
```

Local response. Data processing using Bond objects are usually accomplished by request-reply message pairs. There is a possibility to delegate the processing of some of the messages to the shadow. These cases are: requests for information available at the shadow, denial of some requests, reply to polling etc. These are typical lightweight processing cases. The architecture of Java offer the possibility of caching some of the processing functionality of the object too (we can call this "code caching").

The advantage of local processing is the faster response and lower bandwidth usage.

Security features of shadows. The shadows being the main communication ports in the Bond object, they are the main implementation points of the security features of the system. Shadow can encapsulate securit features by the use of:

-restricted shadows
-message encryption
-message authentification / signed messages

5 Finding objects in the Bond system

Finding an object in the Bond system is equivalent of getting a shadow of it. You may be interested in finding a particular Bond object or a Bond object specified by its properties. A Bond object can be found using the `bondDirectory` object. From the implementation point of view a `bondDirectory` consists of a repository of pointers toward Bond objects and shadows of other (remote) `bondDirectory` objects. In order to realize a distributed directory service, directory objects are arranged in their own virtual network.

Every Bond program has at least one `bondDirectory` object called `dir`. This object is created by the `initBond()` function call, and every locally created Bond object will be automatically registered in it, and unregistered when it is garbage collected.

Finding an object by name can be done using the function:

```
bondShadow find(String name)
```

The following example shows how to access a local object by name using the `dir` object:

```
bondShadow bsh = dir.find("name_of_the_object");
bondObject bo = bsh.realize();
```

Finding an object by its properties is done using the function

`bondShadow find(bondQuery name)`

The following example shows how we can start a scheduler agent on ector:

```
// building a query
bondQuery bq = new bondQuery();
bq.TYPE = "agent";
bq.SUBTYPE = "bondSchedulerAgent";
bq.HOST = "ector";
// finding the object
bondShadow bsh = dir.find(bq); // the program
bondShadow bsh_agent = bsh.start();
// now we can start using the scheduler agent
bsh_agent.say("(tell :content report)");
```

Asyncronous find. The `bondDirectory` being a regular bondObject understands KQML as any other bondObject using say. The results will be obtained by another message.

```
dir.say("(ask content: find querystring: name_of_the_object)", this)
```

The `dir` object will reply by sending a message to the calling object.

If the queried object is local, calling the `find` function is more efficient, because it eliminates the overhead of KQML parsing, which is comparable with the time needed by the directory to find a local object. However, if the object is remote, we can use asyncrhonous find using the message form, in order to prevent the blocking of the requesting object while the request is searched.

The message passing method is used for querying between remote directory objects. In order to find a remote object the directory objects can be put in various configurations, a common one being to link the local `dir` object to a larger object on the Bond server.

6 Appendix: Introduction to KQML

The Knowledge Query and Manipulation Language (KQML) is language for communication between software agents. KQML offers a variety of message

types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests.

The model of message transport of KQML assumes that agents are connected by unidirectional links that carry discrete messages. There may be delays in the message transport, but messages to a single destination arrive in the order they were sent. Message delivery is reliable, but there is no assumption of reliability for the agents.

A KQML message is also called a performative. A performative is expressed as an ASCII string, using a Common Lisp Polish-prefix notation. The first word in the string is the name of the performative, followed by parameters. Parameters in performatives are indexed by keywords and therefore order independent. These keywords, called *parameter names*, must begin with a colon (:) and must precede the corresponding *parameter value*.

One example of a KQML message is the following:

```
(achieve
      :sender ''1002:scheduler@voronet.cs.purdue.edu''
      :receiver ''3453:executor@padis.cs.purdue.edu''
      :content Run
      :reply-with ''mes:5:3453:executor@padis.cs.purdue.edu''
      :bondProgram ''fft''
      :bondInput1 ''ector.cs.purdue.edu/homes/boloni/image.tif''
      :bondOutput1 ''transformed.tif''
)
```

There are 35 reserved performatives but the user can define its own performatives. However if a reserved performative is used, it should be used according to the KQML specification.

Reserved performative parameters

The following parameters are *reserved* in the sense that any performative's use of parameters with those keywords must be consistent with the definitions below.

`:sender` the actual sender of the performative

`:receiver` the actual receiver of the performative

`:from` the origin of the performative in `:content` when *forward* is used

`:to` the final destination of the performative in `:content` when *forward* is used

`:in-reply-to` the expected label in a response to a previous message (same as the `reply-with` value of the previous message)

`:reply-with` the expected label in a response to the current message

`:content` information about which the performative expresses an attitute - in the Bond system, this parameter contents the name of the called remote function or command sent.

Acknowledgments

This work was supported in part by the National Science Foundation through grants BIR-9301210 and MCR-9527131, by a grant from Intel Corporation and by the Scalable I/O Initiative.

References

[1] Ö. Babaoglu, K. Marzullo *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms* Technical Report UBLCS-93-1, January 1993

[2] L. Boloni, K.K. Jun and D.C. Marinescu: *QoS and Reliability Models for Network Computing* Dèpartment of Computer Sciences, Purdue University CSD-TR #97-051

[3] K. Mani Chandy, A. Rifkin, Paolo A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka and L. Weissman *A World-Wide Distributed System Using Java and the Internet* IEEE International Symposion on High Performance Distributed Computing, August 1996.

[4] K. Mani Chandy *Caltech Infospheres Project Overview: Information Infrastructures for Task Forces*

[5] M.Lewis and A. Grimshaw *The Core Legion Object Model* Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996.

[6] R. Orfali, D. Harkey, J. Edwards *Instant CORBA* Wiley Computer Publishing 1996

[7] M. G. Sirbu, D. C. Marinescu: *Bond - A Parallel Virtual Environment* Proceedings of HPCN Europe '96, pp 722-728, Lecture Notes in Computer Science, Volume 1067, Springer Verlag, 1996

[8] M. G. Sirbu, *The Design of a Metacomputing Environment.* Doctoral Thesis, August 1997.

[9] M. G. Sirbu, D. C. Marinescu *A Scheduling Expert Advisor for Heterogeneous Environments* Proceedings of the HCW'97 Heterogeneous Computing Workshop, pp. 74-82, April 1997

[10] D.C. Marinescu *Software Development for Intranet Applications*, Proc DCIA 98, pp 31-50, January 1998

[11] T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, Junc 1993

[12] Y. Labrou, T. Finin *A Proposal for a new KQML Specification* UMBC TR-CS-97-03

[13] P. Hertmann *Illustrated Guide to HTTP*, Manning Publications 1997

[14] W. Grapp, E. Lusk and A. Skjellum *Using MPI*, MIT Press 1994

[15] D. Flanagan *Java in a Nutshell*, O'Reilly & Associates Inc. 1996